Reliability

Terminology

Reliability ::= measure of success with which a system conforms to its specification or a low failure rate.

Failure ::= deviation of a system from its specification

Error ::= system state which lead to failures

Fault ::= the reason for an error

Faults on different levels

- Inconsistent or inadequate specification
  - very frequent source for disastrous faults
- Software design errors
  - very frequent source for disastrous faults
- Component & communication system failures
  - rare and mostly predictable

Faults in the time domain

- Transient faults
  - many communication system failures, electric interference, etc.
- Intermittent faults
  - transient errors which occur more than once (e.g. overheating effects)
- Permanent faults
  - stay in the system until they are repaired by some means

Achieving reliability

System identification

Investigate:

- static applications specifications
- physical sensors and converters constraints
- constraints of the employed controller network
- constraints of the underlying run-time system
- dynamic application specifications (requested real-time behaviour)

Understanding all critical real-time requirements and issues
Fault avoidance

Fault avoidance at hardware-level:
• use reliable hardware components — consider the environmental demands!
• use an adequate hardware system design — shock, humidity, interference, ...
• ensure proper assembly and encapsulation — weak connectors, bad connectors, ...

Fault avoidance at software design level:
• strict system specifications (employ format methods if applicable)
• use proven software-engineering and design methodologies
• employ languages and run-time environments with reasonable support for the requirements.

Find and remove errors from the previous stage.
• re-utilization method indicates the absence of faults
• even formal methods cannot identify specification faults
• and specifically for real-time and embedded systems
• other tests cannot be performed under realistic conditions
• most simulation environments have a severe impact on real-time systems
• the test space for real-time system is significantly larger than for non-real-time systems

Regardless of the rigor of fault prevention methods:
• the real-time system might still fail
This is specifically critical for all non-monitored systems:
• systems which are (temporarily) inaccessible
• systems which are (temporary) inaccessible
• un-manned vehicles which operate autonomously by default
• systems in remote / dangerous environments

Instead (or in addition to fault prevention): enabling a ‘safe landing’:
☞ Fault tolerance

Fault tolerance

• Full fault tolerance — the system continues to operate in the presence of foreseeable error conditions
• without any significant failures — also this might induce a reduced operation period.
• Graceful degradation (fail soft) — the system continues to operate in the presence of foreseeable error conditions, accepting a partial loss of functionality or performance.
• Fail safe — the system halts and maintains its integrity.

Fault removal

• for the detection of failures and the localization of faults
• for the handling of exceptional situations and error-recovery.
• as a functional duplication or multiplication of complete (sub-)systems in order to hot-swap or select the operational one in case of a failure in one part of the (sub-)system.
• Fault-detection and recovery hardware includes: watch-dog timers, limit switches, additional physical sensors, transient-recording-systems (emergency system dump), overload-backup systems, or even in-circuit simulators.
• Triple Modular Redundancy (TMR) or N-Modular Redundancy (NMR) assumes: functionally identical components which are either:
  • static parts of the system and connected via a voting/marking/comparing system
  • or in case of a detected error-condition: dynamic parts which are swapped in.

Hardware redundancy

• any hardware redundancy adds to the overall system complexity!
In case of TMR or NMR:
• the assumption that an error occurs in one part of the system only requires that either:
  • the fault is based on a physical phenomenon, which applies only locally
  • or the structure of the functionally identical systems is sufficiently different

For some high-risk systems this approach is applied in forms of redundant sub-systems with:
• the same specification
• different computer systems (CPUs, buses, memory systems, drives)
• different operating systems
• different real-time languages and development environments (N-Version programming)
• and by restricting the communication between the different developer teams

Not too surprisingly, the outputs from the different systems are slightly different ...

Fault avoidance

Fault avoidance at software design level:
• use reliable hardware components — consider the environmental demands!
• use an adequate hardware system design — shock, humidity, interference, ...
• ensure proper assembly and encapsulation — weak connectors, bad connectors, ...

Fault avoidance at software design level:
• strict system specifications (employ format methods if applicable)
• use proven software-engineering and design methodologies
• employ languages and run-time environments with reasonable support for the requirements.

Targeted failure probability: < 10\(^{-10}\)/h (e.g. UK Sizewell B nuclear reactor (emerg.): < 10\(^{-7}\)/h)

No single fault on board the 777 should cause more than the loss of one primary flight computer.

3 identical primary flight computers distributed in the Boeing 777, each consisting of:
• 3 processors: AMD 29050, Motorola 68040, INTEL 80486 (called ‘lanes’)
• independent power-sources and inertia measurements
• code build by 3 different Ada compilers

3 identical primary flight computers distributed in the Boeing 777, each consisting of:
• 3 processors: AMD 29050, Motorola 68040, INTEL 80486 (called ‘lanes’)
• independent power-sources and inertia measurements
• code build by 3 different Ada compilers

Native Ada source code (the specification): around 3 million lines of code, but different monitor functions

N-version programming

Impacts to software diversity:

<table>
<thead>
<tr>
<th>Development team</th>
<th>Languages</th>
<th>Tools</th>
<th>Algorithms</th>
<th>Methodologies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifications</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coding</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Not a single fatal event — information from November 2001

Fault removal

Fault removal

Fault prevention

(avoidance & removal)

This is specifically critical for all non-monitored systems:
• systems which are (temporarily) inaccessible
• un-manned vehicles which operate autonomously by default
• systems in remote / dangerous environments

Instead (or in addition to fault prevention): enabling a ‘safe landing’:
☞ Fault tolerance

Fault avoidance

Fault avoidance at software design level:
• strict system specifications (employ format methods if applicable)
• use proven software-engineering and design methodologies
• employ languages and run-time environments with reasonable support for the requirements.

Fault avoidance at hardware-level:
• use reliable hardware components — consider the environmental demands!
• use an adequate hardware system design — shock, humidity, interference, ...
• ensure proper assembly and encapsulation — weak connectors, bad connectors, ...

Fault avoidance at software design level:
• strict system specifications (employ format methods if applicable)
• use proven software-engineering and design methodologies
• employ languages and run-time environments with reasonable support for the requirements.

Fault avoidance at hardware-level:
• use reliable hardware components — consider the environmental demands!
• use an adequate hardware system design — shock, humidity, interference, ...
• ensure proper assembly and encapsulation — weak connectors, bad connectors, ...

Fault avoidance at software design level:
• strict system specifications (employ format methods if applicable)
• use proven software-engineering and design methodologies
• employ languages and run-time environments with reasonable support for the requirements.

Fault avoidance at hardware-level:
• use reliable hardware components — consider the environmental demands!
• use an adequate hardware system design — shock, humidity, interference, ...
• ensure proper assembly and encapsulation — weak connectors, bad connectors, ...

Fault avoidance at software design level:
• strict system specifications (employ format methods if applicable)
• use proven software-engineering and design methodologies
• employ languages and run-time environments with reasonable support for the requirements.

Fault avoidance at hardware-level:
• use reliable hardware components — consider the environmental demands!
• use an adequate hardware system design — shock, humidity, interference, ...
• ensure proper assembly and encapsulation — weak connectors, bad connectors, ...

Fault avoidance at software design level:
• strict system specifications (employ format methods if applicable)
• use proven software-engineering and design methodologies
• employ languages and run-time environments with reasonable support for the requirements.

Fault avoidance at hardware-level:
• use reliable hardware components — consider the environmental demands!
• use an adequate hardware system design — shock, humidity, interference, ...
• ensure proper assembly and encapsulation — weak connectors, bad connectors, ...

Fault avoidance at software design level:
• strict system specifications (employ format methods if applicable)
• use proven software-engineering and design methodologies
• employ languages and run-time environments with reasonable support for the requirements.

Fault avoidance at hardware-level:
• use reliable hardware components — consider the environmental demands!
• use an adequate hardware system design — shock, humidity, interference, ...
• ensure proper assembly and encapsulation — weak connectors, bad connectors, ...

Fault avoidance at software design level:
• strict system specifications (employ format methods if applicable)
• use proven software-engineering and design methodologies
• employ languages and run-time environments with reasonable support for the requirements.

Fault avoidance at hardware-level:
• use reliable hardware components — consider the environmental demands!
• use an adequate hardware system design — shock, humidity, interference, ...
• ensure proper assembly and encapsulation — weak connectors, bad connectors, ...

Fault avoidance at software design level:
• strict system specifications (employ format methods if applicable)
• use proven software-engineering and design methodologies
• employ languages and run-time environments with reasonable support for the requirements.

Fault avoidance at hardware-level:
• use reliable hardware components — consider the environmental demands!
• use an adequate hardware system design — shock, humidity, interference, ...
• ensure proper assembly and encapsulation — weak connectors, bad connectors, ...

Fault avoidance at software design level:
• strict system specifications (employ format methods if applicable)
• use proven software-engineering and design methodologies
• employ languages and run-time environments with reasonable support for the requirements.

Fault avoidance at hardware-level:
• use reliable hardware components — consider the environmental demands!
• use an adequate hardware system design — shock, humidity, interference, ...
• ensure proper assembly and encapsulation — weak connectors, bad connectors, ...

Fault avoidance at software design level:
• strict system specifications (employ format methods if applicable)
• use proven software-engineering and design methodologies
• employ languages and run-time environments with reasonable support for the requirements.

Fault avoidance at hardware-level:
• use reliable hardware components — consider the environmental demands!
• use an adequate hardware system design — shock, humidity, interference, ...
• ensure proper assembly and encapsulation — weak connectors, bad connectors, ...

Fault avoidance at software design level:
• strict system specifications (employ format methods if applicable)
• use proven software-engineering and design methodologies
• employ languages and run-time environments with reasonable support for the requirements.

Fault avoidance at hardware-level:
• use reliable hardware components — consider the environmental demands!
• use an adequate hardware system design — shock, humidity, interference, ...
• ensure proper assembly and encapsulation — weak connectors, bad connectors, ...

Fault avoidance at software design level:
• strict system specifications (employ format methods if applicable)
• use proven software-engineering and design methodologies
• employ languages and run-time environments with reasonable support for the requirements.

Fault avoidance at hardware-level:
• use reliable hardware components — consider the environmental demands!
• use an adequate hardware system design — shock, humidity, interference, ...
• ensure proper assembly and encapsulation — weak connectors, bad connectors, ...

Fault avoidance at software design level:
• strict system specifications (employ format methods if applicable)
• use proven software-engineering and design methodologies
• employ languages and run-time environments with reasonable support for the requirements.
A very well structured system is the cornerstone of damage diagnosis. This involves complex mode and priority changes (deadlines might still be relevant).

- Method of choice for most time critical parts of real-time and embedded systems.
- Fault treatment: In order to prevent the same error state again, the fault itself might/should be eliminated. If a error state is detected: set back to the last consistent checkpoint.
- In general: diverging results do not necessarily imply any faults.

**Dynamic redundancy** — Error diagnosis

**Error states** from the environment:
- Hardware: CPU, controllers, communication systems, ...
- Run-time environment
- Error states stemming from checks which do not affect the application processes:
  - Replication
  - Timing
  - Reversal
  - Coding
  - Reasonableness: check assertions (e.g. in Eiffel)
  - Structural: check structural integrity (e.g. lists, file-systems)
  - Continuity: assuming a limited difference between consecutive controller values.

**Confinement**:

- **How to avoid the transfer of fault-effects between system parts?**
  - Modular decomposition
  - Atomic actions
  - Firewalls

**Assessment**:

- resulting from the location of the detected error state and the possible paths though the system which are all leading to this error state.
- A time-granular system structure (error-confinement) limits the length of these possible paths.

- A very well structured system is the cornerstone of damage diagnosis.

**Dynamic redundancy** — Error recovery

Backward error recovery:
- set checkpoints and save the system state with each passing of a checkpoint.
- How can system-wide consistent checkpoints be ensured?
- if a error state is detected: set back to the last consistent checkpoint.
- applicable even if the fault itself can not be identified.
- not applicable at all, if the system contains non-reversible or -resetable components (time, ...)

**Forward error recovery**: method of choice for most time critical parts of real-time and embedded systems.
- Highly application dependent.
- May involve complex mode and priority changes (deadlines might be still relevant).

**N-version programming**  

**Voting issues**

- Integer arithmetic:
  - Integer (or any discrete sub-type) based results will be identical.
- Real arithmetic:
  - Real-valued results will usually be different.
  - Comparisons need to consider tolerances.
  - If the process is not fully continuous (thresholds, quantizations, bifurcations)
    - Comparisons need to re-model the whole process in order to evaluate similarities

- Independence: re-specify the system

- Multiple solutions:
  - The solution space itself allows for multiple correct, but different solutions

**Real-valued results will usually be different**.

**Comparisons need to consider tolerances.**

- If the process is not fully continuous (thresholds, quantizations, bifurcations)
  - Comparisons need to re-model the whole process in order to evaluate similarities

- Independence: re-specify the system

- Multiple solutions: The solution space itself allows for multiple correct, but different solutions

<table>
<thead>
<tr>
<th>Language</th>
<th>Sources (l.o.c.)</th>
<th>Test runs</th>
<th>Errors</th>
<th>Failure-rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>2256</td>
<td>512000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>'C'</td>
<td>1531</td>
<td>568</td>
<td>1.100×10⁻⁸</td>
<td></td>
</tr>
<tr>
<td>Modula-2</td>
<td>1562</td>
<td>&quot;</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pascal</td>
<td>2331</td>
<td>&quot;</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Prolog</td>
<td>2228</td>
<td>&quot;</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T (close to Lisp)</td>
<td>1568</td>
<td>&quot;</td>
<td>600</td>
<td>1.320×10⁻³</td>
</tr>
<tr>
<td>Average</td>
<td>1993</td>
<td>&quot;</td>
<td>321</td>
<td>0.627×10⁻³</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Failure category</th>
<th>Average single version failure probabilities (1,527,400 cases)</th>
<th>Average 3-version failure probabilities (1,023,480,000 cases)</th>
<th>Average 5-version failure probabilities (3,076,400 cases)</th>
</tr>
</thead>
<tbody>
<tr>
<td>no errors</td>
<td>0.999931733</td>
<td>0.9998409</td>
<td>0.997887</td>
</tr>
<tr>
<td>single error</td>
<td>6.27×10⁻³</td>
<td>1.38×10⁻²</td>
<td>1.9×10⁻²</td>
</tr>
<tr>
<td>two distinct errors</td>
<td>0.2×10⁻²</td>
<td>0.2×10⁻²</td>
<td>0.2×10⁻²</td>
</tr>
<tr>
<td>two coincident errors</td>
<td>0.2×10⁻²</td>
<td>0.2×10⁻²</td>
<td>0.2×10⁻²</td>
</tr>
<tr>
<td>three errors</td>
<td>0.5×10⁻³</td>
<td>0.5×10⁻³</td>
<td>0.5×10⁻³</td>
</tr>
</tbody>
</table>
assuming that there is a sequence of distinguishable states (or ‘time’).}

Collission Warning 

Collission Avoidance

true

AAAA

true

AA

-e.g.

- are there any safe and functional systems beyond a certain complexity? ... aeroplanes? cars?

- Dependability:
  - Availability — ready to use
  - Reliability — absence of failures
  - Safety — absence of fatal failures
  - Confidentiality — absence of unauthorized disclosures
  - Integrity — no data corruptions
  - Maintainability — accessibility to changes and improvements

- Adding the concepts of ordering for events and states
- Suitable for event driven systems, reactive systems — required if interrupts are to be handled.
  - Protected procedures as interrupt handlers

- Extending predicate logic
- Assertions on sequences and orders of states — introduces a dynamic feature into the run-time that has complexity and overhead.

- Temporal logic
- Ada95 Ravenscar profile (Burns, Dobbing, Romanski ‘98)
  - Task type and object declarations at the library level — no hierarchy of tasks, and hence no exit protocols needed from blocks and sub-programs.
  - No dynamic allocation or unchecked deallocation of protected and task objects — no use of task entries
  - Tasks are assumed to be non-terminating — the absolute form of delay is the correct one to use for constructing periodic tasks.
  - Library level Protected objects with no entries — “Delay until” statement but no “delay” statement
  - Library level Protected objects with a single entry — the need to protect data structures against asynchronous task actions.
  - No Abort or ATC — these features leads to the greatest overhead in the run-time system due to the need to protect data structures against asynchronous task actions.
  - No No Calendar package — cannot be used for some algorithms and has low overhead.
  - Ada.Task_Identification — can be useful for some algorithms and has low overhead, available in reduced form (no Abort_Task or task attribute functions Callable or Terminated).
  - Atomic and Volatile pragmas — introduces a dynamic feature into the run-time that has complexity and overhead.

- Ada95 Ravenscar profile (Burns, Dobbing, Romanski ‘98)
  - Barrier consisting of a single boolean variable — no side effects are possible and exit protocol becomes simple.
  - Only a single task may queue on an entry — hence no queue required; this is a static property that can easily be verified, or it can lead to a bounded error at runtime.
  - No No Calendar package — “Real-Time” package is sufficient.
  - No use of the select statement — non-deterministic behaviour is difficult to analyse, moreover the existence of protected objects has diminished the importance of the select statement to the task model.
  - No use of task entries — not necessary to program systems that can be analysed; it follows that there is no need for the accept statement.

- Ada95 Ravenscar profile (Burns, Dobbing, Romanski ‘98)
  - “Delay until” statement but no “delay” statement — the absolute form of delay is the correct one to use for constructing periodic tasks.
  - “Real-Time” package — to gain access to the real-time clock.
  - No Calendar package — “Real-Time” package is sufficient.
  - Atomic and Volatile pragmas — needed to enforce the correct use of shared data.
  - Count attribute (but not within entry barriers) — can be useful for some algorithms and has low overhead.
  - Ada.Task_Identification — can be useful for some algorithms and has low overhead, available in reduced form (no Abort_Task or task attribute functions Callable or Terminated).
  - Task discriminants — can be useful for some algorithms and has low overhead.

- Granularity is usually finer than in static redundant systems.

- Exchange of faulty components is nevertheless usually an expensive and complex operation.

- the number of substitutable sub-systems in a dynamic redundant system is still limited.

(many systems will assume transient faults, log the event and continue operations ...)
Temporal logic

- Another temporal operator:
  \( \text{ApB} \): A holds until the first occurrence of B, which will occur eventually.
  
  e.g. \( ((\text{Tasks}_\text{Waiting} \mu \text{Entry}_\text{Closed}) \land \neg (\text{Tasks}_\text{Waiting} \mu \text{Entry}_\text{Open})) \)

- Temporal logic expresses the order of events only and has means to express temporal scopes, deadlines, ...

### Linear Temporal Logic of Real Numbers (LTR)

\[
\phi := \neg p \lor \exists t. U \phi \lor \exists t. S \phi
\]

where

\[
\begin{align*}
(t, t') & > p \quad \text{iff } \exists t \in (t, t') \exists \phi \lor \exists t \in (t, t') \exists \phi
\\
(t, t') & > 0 \lor \exists t \lor \exists t' \lor \exists t \lor \exists t' \lor \exists t' \\
\phi & \text{ is satisfiable iff } \exists (t, t') \lor \phi \quad \text{is valid iff } \forall (t, t') \lor \phi
\end{align*}
\]

### Event-Clock Temporal Logic

\[
\phi := \neg p \lor i \lor \exists t \lor \exists \phi \lor \exists t \lor \exists \phi
\]

where

\[
\begin{align*}
(t, t') & > p \quad \text{iff } \exists t \lor \exists t' \land \phi
\\
(t, t') & > 0 \lor \exists t \lor \exists t' \lor \exists t \lor \exists t' \lor \exists t' \\
\phi & \text{ is satisfiable iff } \exists (t, t') \lor \phi \quad \text{is valid iff } \forall (t, t') \lor \phi
\end{align*}
\]

### Metric-Interval Temporal Logic

\[
\phi := \neg p \lor \exists t \lor \exists \phi \lor \exists t \lor \exists \phi
\]

where

\[
\begin{align*}
(t, t') & > p \quad \text{iff } \exists t \lor \exists t' \land \phi
\\
(t, t') & > 0 \lor \exists t \lor \exists t' \lor \exists t \lor \exists t' \lor \exists t' \\
\phi & \text{ is satisfiable iff } \exists (t, t') \lor \phi \quad \text{is valid iff } \forall (t, t') \lor \phi
\end{align*}
\]